Result Certification of Static Program Analysers with Automated Theorem Provers*

Frédéric Besson, Pierre-Emmanuel Cornilleau, and Thomas Jensen

Inria Rennes – Bretagne Atlantique
Campus de Beaulieu, 35042, Rennes Cedex, France
firstname.lastname@inria.fr

Abstract. The automation of the deductive approach to program verification crucially depends on the ability to efficiently infer and discharge program invariants. In an ideal world, user-provided invariants would be strengthened by incorporating the result of static analysers as untrusted annotations and discharged by automated theorem provers. However, the results of object-oriented analyses are heavily quantified and cannot be discharged, within reasonable time limits, by state-of-the-art automated theorem provers. In the present work, we investigate an original approach for verifying automatically and efficiently the result of certain classes of object-oriented static analyses using off-the-shelf automated theorem provers. We propose to generate verification conditions that are generic enough to capture, not a single, but a *family* of analyses which encompasses Java bytecode verification and Fähndrich and Leino type-system for checking null pointers. For those analyses, we show how to generate tractable verification conditions that are still quantified but fall in a decidable logic fragment that is reducible to the Effectively Propositional logic. Our experiments confirm that such verification conditions are efficiently discharged by off-the-shelf automated theorem provers.

1 Introduction

In recent years, the automation of deductive program verification frameworks (*e.g.*, [5,15,26,10]) has made impressive progress. Proving the functional correctness of real programs can now be done with reasonable effort. A major automation breakthrough is due to the improvements of automated theorem provers (ATPs) (notably Satisfiability Modulo Theory (SMT) solvers [18,12,6]) that allow to routinely and efficiently discharge first-order verification conditions. At the same time, static analysers have also made significant progress. They can infer automatically sophisticated invariants that would strengthen user-provided invariants and therefore automate further the verification process. Yet, this potential for further automation has not fully materialised yet. Indeed, there are still obstacles hindering the systematic integration of static analysis results into deductive program verification frameworks.

* This work was partly funded by the ANR DeCert and FNRAE ASCERT projects.

There are two main approaches for integrating automatically generated invariants into deductive program verification frameworks depending on whether static analyses results are *trusted* or *untrusted*. Static analyses that are trusted are usually built into the verification methodology. For instance, SPEC♯ [5] is using `@NonNull` type information [20] to generate Boogie [4] intermediate code and the Why3 platform [10] is using an effect system to tame aliasing. In this scenario, those static analyses are part of the Trusted Computing Base (TCB). Hence, the addition of a novel analysis is potentially jeopardising the soundness of whole verification methodology. In another approach, static analyses results are untrusted and are treated as candidate invariants which, following the verification process, are transformed into verification conditions that are eventually discharged by automated theorem provers. For instance, candidate invariants generated by HOUDINI [21] are validated by ESC/JAVA [22]. This integration scheme has the advantage that static analyses are not part of the TCB and therefore an error in the static analysis, or a misinterpretation of the static analysis result, cannot compromise the soundness of the verification methodology.

This latter approach comes with both theoretical and practical challenges. If the static analysis and the verification methodology are grounded on semantics that are too far apart, filling the semantic gap may prove unfeasible or be responsible for an unbearable encoding overhead. Semantic discrepancies can show up in multiple places. A typical example is the modelling of machine integers in case of overflows: is it an error or a normal behaviour? More serious is the question of the memory models that can be incompatible, especially if the verification methodology enforces a hardwired alias control mechanism or object ownership. In the propitious case that the analysis result can be encoded in logical form with reasonable overhead, there is absolutely no guarantee that the verification conditions will be automatically discharged by automated theorem provers. Because static analyses operate over a program logic that is essentially computable, the loss of decidability comes from the logic encoding of the static analysis result. This absence of (relative) completeness of ATPs *w.r.t.* a particular static analysis makes this approach for validating static analyses fragile and unpredictable.

This paper aims at ensuring that proof obligations originating from static analysis results can be discharged with *certainty* by automated theorem provers. The result certification of static analyses has been studied for its own sake. However, existing works propose *ad hoc* solutions that are specific to a single analysis *e.g.,* for polyhedral program analyses [9] and register allocation in the CompCert C compiler [29]. A universal solution, working for arbitrary analyses, is very likely impossible. We propose an intermediate solution which leverages the deductive power of automated provers and covers a relevant *family* of static analyses for object-oriented languages. For this family, we show how to generate tractable verification conditions that are reducible to the Effectively Propositional (EPR) fragment of first-order logic. This fragment is decidable and in practise state-of-the-art automated provers are able of discharge the proof obligations.

We shall consider static analyses that have been defined using the theory of abstract interpretation. The first step in the result verification consists of translating the elements of the analyser's abstract domain into a logical formalism in which the semantic correctness of the analysis can be expressed. Our translation is defined using the concretisation function γ of the abstract interpretation which maps abstract domain elements to properties of the concrete semantic domain. More precisely, it translates abstract domain elements associated to program points into pre- and post-conditions, expressed in Many Sorted First Order Logic (MSFOL). Running a Verification Condition Generator (VCGen) on such an annotated program results in a set of proof obligations expressed in first-order logic. These Verification Conditions (VCs) are then given to be proved by ATPs. As already mentioned, this is no formal guarantee that ATPs will be able to discharge those proof obligations. For object-oriented analyses, the formulae make extensive use of quantifiers and are therefore challenging for ATPs. This very work is motivated by the experimental observation that state-of-the-art ATPs are in practise incapable of discharging those formulae. An important part of this paper is therefore concerned with identifying a logical fragment for expressing pre-, post-conditions and VCs, and to present a method for transforming these VCs into VCs that can be discharged efficiently.

1.1 Overview

Our approach to get tractable verification conditions is to restrict our attention to a family of object-oriented static analyses. Each static analysis in the family is equipped with a specific *base* abstract domain and is thus equipped with a specific concretisation function. Yet, the lifting of this analysis specific concretisation function to the program heap is generic and shared by all the analyses in the family. We exploit these similarities to generate specialised verification conditions that are reducible to EPR. To demonstrate the approach, we have developed result certifiers for two different object-oriented static analyses belonging to the family: a bytecode verifier (BCV) [28] for Java and Fähndrich and Leino type-system [20] for checking null pointers.

We restrict ourselves to static analyses based on the theory of abstract interpretation [17]. In this framework static analyses are defined *w.r.t.* a *collecting semantics* which extracts the properties of interest from the program concrete semantics. For formalisation purposes, we give a core object-oriented bytecode language (see Section 2) a mostly small-step operational semantics $\cdot \rightarrow \cdot \subseteq \mathcal{State} \times \mathcal{State}$, *i.e.*, a small-step semantics with big-step reduction for method calls. A program state $(e, h, p) \in \mathcal{Env} \times \mathcal{Heap} \times \mathcal{PP}$ is a triple where e is a local environment mapping variables to locations; h is a mapping from locations to objects representing the heap and p is the current program point. The semantics is fairly standard except for a generic instrumentation of instance fields. This instrumentation is expressed using a dedicated $\mathcal{IF}$ domain with a specific element *inull* and an operation *ifield* that models field update.

$$\mathit{inull} : \mathcal{IF} \qquad \mathit{ifield} : \mathcal{IF} \rightarrow \mathcal{IF}$$

Fields of a newly created object are tagged by *inull* and the *ifield* function is called whenever a field is updated. Using the terminology of deductive verification, for each field, we add a ghost field that is updated together with the concrete field. By construction, the instrumentation is transparent *i.e.*, erasing the instrumentation has no impact on the semantics.

Each static analysis is defined by a particular instrumentation $(\mathcal{IF}, \mathit{inull}, \mathit{ifield})$ and by an abstract domain Abs equipped with a concretisation function:

$$\gamma : \mathit{Abs} \to \mathcal{P}(\mathcal{State})$$

As collecting semantics we consider the set of reachable *instrumented* states $\mathcal{Reach}$ of the program semantics. A correct (over-)approximation of $\mathcal{Reach}$ is an abstract element $b^\sharp \in \mathit{Abs}$ whose concretisation is such that $\mathcal{Reach} \subseteq \gamma(b^\sharp)$. As a result, verifying the static analysis result amounts to proving the following proof obligation:

$$s \in \gamma(b^\sharp) \wedge s \to s' \Rightarrow s' \in \gamma(b^\sharp)$$

Providing the concretisation function and the program semantics can be axiomatised in first-order logic, the proof obligation can be sent to ATPs such as SMT solvers or first-order provers. In our case, the logic embedding does not incur a particular encoding overhead as it is demonstrated by our modelling of the semantics [7] using the Why platform.

This approach has demonstrated its effectiveness to certify the result of numerical analyses [16]. However, for object-oriented analyses, ATPs fail to discharge the proof obligation because the formulae quantify over infinite domains such as the set of memory locations. An obvious optimisation that simplifies the task of the prover consists in splitting the proof obligation into program point specific verification conditions. In our experiments, off-the-shelf ATPs still fail to reliably and consistently discharge all the proof obligations. This absence of (relative) completeness of ATPs *w.r.t.* a particular static analysis makes this approach for validating static analyses fragile and unpredictable.

To alleviate the problem, provers could be tuned on a *per* analysis basis. However, this solution is fragile and comes without any formal guarantee: what about its robustness *w.r.t.* slight modifications of the static analysis? Here, we explore another solution that is robust and does not require any modification of the provers. We propose to tame static analyses so that, by construction, proof obligations fall in fragments that are well-understood by the prover and are therefore discharged reliably. The *family* of static analyses we have identified can be characterised almost syntactically by the definition of concretisation function. As a result, the static analysis designer can have the guarantee that the proof obligations will be discharged without any knowledge of the internals of the provers. The proof obligations we generate are easily reducible to Effectively Propositional logic. This logic is still quantified but decidable and, as the ATP System Competition shows, existing provers are already tuned for this logic.

Our family of analyses is parametrised by a base abstract domain $\mathcal{Val}^\sharp$ for abstracting values (see Section 3). This abstract domain is automatically lifted

to the program abstraction Abs.

$$\mathcal{Heap}^\sharp = \mathcal{F} \to \mathcal{Val}^\sharp \times \mathcal{Val}^\sharp \qquad \mathcal{Env}^\sharp = \mathcal{Var} \to \mathcal{Val}^\sharp \quad Abs = \mathcal{Heap}^\sharp \times (\mathcal{PP} \to \mathcal{Env}^\sharp)$$

This abstraction corresponds to static analyses that are flow-sensitive (the abstraction of local variables is program specific) and heap-insensitive (the abstraction of the heap is shared by every program-point). The abstraction of fields is relational: a field f is abstracted by a pair $(v_1, v_2) \in \mathcal{Val}^\sharp \times \mathcal{Val}^\sharp$ such that v_2 is the abstraction of `x.f` providing v_1 is the abstraction of `x`. This family of analyses encompasses well-known static analyses such as Java bytecode verification [28] and Fähndrich and Leino type-system for checking null pointers [20].

For this family of analyses, we show how to generate verification conditions of the following shape (see Section 4.1):

$$\forall \bar{c} \in \mathcal{Class}, \bar{f} \in \mathcal{F}, \bar{i} \in \mathcal{IF}, \bar{v} \in \mathcal{Var}.\phi$$

where $\bar{c}, \bar{f}, \bar{i}, \bar{v}$ are vectors of universally quantified variables and ϕ is a quantifier-free propositional formula built over the following atomic propositions

$$p ::= v^\sharp \in \gamma_{null} \mid (c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp) \mid c \preceq c \mid f \in c.$$

In this definition, γ_{null} and $\gamma_{\mathcal{L}}$ are used to specify the concretisation of the base domain $\mathcal{Val}^\sharp$, $\preceq$ is a subclass test and $f \in c$ checks whether a field belongs to a class. As soon as the domain $\mathcal{IF}$ is finite, quantifications are only over finite domains. Thus, the formulae are Effectively propositional providing that $\gamma_{\mathcal{L}}$ can be expressed in this fragment.

Using our approach, the designer of the analysis can verify at the analysis level the logic fragment the verification conditions will fall into. This is a formal guarantee that makes this technique for verifying the analyses results very robust. In practise, even if the formulae are decidable, the provers might not be complete. Our experiments show that for EPR the provers are really efficient at discharging our verification conditions.

1.2 Organisation

In Section 2 we present a small object-oriented language with its operational semantics. Section defines the family of analyses we consider and presents the encoding of Java bytecode verification and Fähndrich and Leino type-system for checking null pointers [20]. For this specific class of analyses, Section 4 shows how to generate verification conditions in the EPR fragment. A prototype implementation in Why3 is described in Section 5. Section 6 reviews related work and Section 7 concludes.

2 Language, Syntax and Semantics

In the formalisation, we consider a core object-oriented language. Let $\mathcal{PP}$, $\mathcal{Var}$, $\mathcal{Class}$, $\mathcal{Method}$ and $\mathcal{F}$ be finite sets of program points, variable names, classes,

method names and field names. The set $\mathcal{V}ar$ contains distinguished elements for the `this` pointer, the parameters $\mathtt{p}_0$ and $\mathtt{p}_1$ and the method result `res`.

$$\begin{array}{l}\mathcal{V}ar \ni x \ ::= \mathtt{this} \mid \mathtt{p}_0 \mid \mathtt{p}_1 \mid \mathtt{res} \mid \ldots \\ \mathcal{S}tmt \ni s ::= \mathtt{x} := \mathtt{null} \mid \mathtt{x} := \mathtt{y} \mid \mathtt{x} := \mathtt{y.f} \mid \mathtt{x} := \mathtt{new\ C} \mid \mathtt{x} := \mathtt{y.c.m(x,y)} \\ \qquad\qquad \mid \mathtt{x.f} := \mathtt{y} \mid \mathtt{Ifnull(x,pc)} \mid \mathtt{skip}\end{array}$$

Programs: In our model, a method $(\mathtt{c},\mathtt{m}) \in \mathcal{C}lass \times \mathcal{M}ethod$ is identified by its defining class `c` and its name `m`. Its entry point (written $(\mathtt{c.m})_0$) and its exit point (written $(\mathtt{c.m})_\infty$) are given by the mapping $\mathtt{sig} \in \mathcal{C}lass \times \mathcal{M}ethod \to (\mathcal{PP} \times \mathcal{PP})_\perp$. The code is described via two functions: $\mathtt{get\_stmt} \in \mathcal{PP} \to \mathcal{S}tmt$ returns the statement at program point; the normal successor of a program point p is written p^+. For a conditional statement $\mathtt{Ifnull}(e, p')$, if e is null, the successor is p', otherwise it is p^+. The class hierarchy is represented by a relation `extends` relating a class and its direct super-class. The subclass relation $\preceq$ is defined as the reflexive, transitive closure of `extends`. Each class defines a set of fields. We write $f \in c$ for a field that is either defined or inherited by c, *i.e.,* recursively defined by a super-class of c. The `lookup` function models virtual method dispatch and is defined if a matching method is found by walking-up the class hierarchy. We identify a method signature by a defining class and a method name.

$$\mathtt{lookup} : \mathcal{C}lass \to (\mathcal{C}lass \times \mathcal{M}ethod) \to \mathcal{C}lass_\perp$$

Semantics: The semantic domains are built upon an infinite set of locations $\mathcal{L}$, and parametrised by an unspecified domain $\mathcal{IF}$ of field annotations. At object creation, field annotations are tagged by $inull \in \mathcal{IF}$ and updated at field updates by the $ifield : \mathcal{IF} \to \mathcal{IF}$ function. Values are either a location or the constant *null*; an environment is a mapping from variables to values; an object is a pair made of a class and a mapping from fields to values and annotations of fields; the heap is a partial mapping from locations to objects. A state is a tuple of $\mathcal{E}nv \times \mathcal{H}eap \times \mathcal{PP}$. Given a state s, we have $s = (s.env, s.hp, s.cpp)$. We add a set of error states $\mathcal{E}rr$ for null pointer dereferencing and calls to undefined methods or lookup failure.

$$\begin{array}{cccc}\mathcal{V}al = \mathcal{L} \cup \{null\} & \mathcal{E}nv = \mathcal{V}ar \to \mathcal{V}al & \mathcal{O}bj = \mathcal{C}lass \times (\mathcal{F} \to \mathcal{V}al \times \mathcal{IF}) \\ \mathcal{H}eap = \mathcal{L} \to \mathcal{O}bj_\perp & \mathcal{S}tate = \mathcal{E}nv \times \mathcal{H}eap \times \mathcal{PP} & \mathcal{E}rr = \{NullPointer, LookupFail\}\end{array}$$

The semantics rules are given in Fig. 2 of Appendix A. We use a *mostly small-step* presentation of the semantics, defining inductively a relation $\to$ between successive states in the same method and modelling method calls by the transitive closure $\to^*$. The rules for modelling method calls are given below

$$\text{SCall}\ \frac{\begin{array}{c}s.env[y] = l \quad s.hp[l] = (c, o) \quad s.env[a_0] = v_0 \quad s.env[a_1] = v_1 \\ \mathtt{lookup}(c)(c_0, m) = c' \qquad \mathtt{sig}(c', m) = (p_{beg}, p_{end}) \\ env' = (\lambda x.null)[\mathtt{this} \leftarrow l][\mathtt{p}_0 \leftarrow v_0][\mathtt{p}_1 \leftarrow v_1]\end{array}}{s \rhd_{c_0,m} ((env'\ ,\ s.hp\ ,\ p_{beg}), p_{end})}$$

$$\text{Call}\ \frac{\begin{array}{c}\mathtt{get\_stmt}(s.cpp) = \mathtt{x} := \mathtt{y.c_0.m(a_0, a_1)} \quad x\ is\ assignable \\ s \rhd_{c_0,m} (init, p_{end}) \qquad init \longrightarrow^* end \qquad end.cpp = p_{end}\end{array}}{s \longrightarrow (s.env[x \leftarrow end.env[\mathtt{res}]], end.hp, s.cpp^+)}$$

The side-condition *x is assignable* means that $x \notin \{\texttt{this}, \texttt{p}_0, \texttt{p}_1\}$ and ensures that those variables are not mutable. The rule [SCall] is responsible for initialising the environment of a called method and retrieving the method exit point. The rule [Call] models method invocation.

Figure 3 of Appendix A. describes the semantics of programs "which go wrong". The semantics is blocking with respect to ill-formed programs (assignment to the variables `this`, $\texttt{p}_0$ and $\texttt{p}_1$), but programs leading to null pointer dereferencing or *method not found* lead to special error states (*NullPointer* and *LookupFail*).

The *set of reachable states* are obtained by the reflexive, transitive closure of the relation $\twoheadrightarrow$ which enriches the semantic relation $\rightarrow$ with states reachable from sub-calls.

$$\frac{s \rightarrow s'}{s \twoheadrightarrow s'} \qquad \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \quad s \rhd_{c_0,m} (s', p_{end})}{s \twoheadrightarrow s'}$$

The set of reachable states for an initial set S_0 of initial states is then defined as

$$\mathit{Reach} = \{s \mid s_0 \in S_0 \wedge s_0 \twoheadrightarrow^* s\}.$$

The role of static analyses presented in the next section is to compute an abstract over-approximation Abs of the set of reachable states ($\mathit{Reach} \subseteq \gamma(Abs)$) that can rule out the run-time errors *NullPointer* and *LookupFail*.

3 Defining a Family of Analyses

To obtain a more parsimonious embedding of abstract domains into pre- and post-conditions, we restrict ourselves to a particular class of analyses. This class is defined by a parametrisation of the operational semantics and of the concretisation of the analyses it contains.

3.1 Parametrised Analyses

We restrict our attention to analyses parametrised by a particular instrumentation of the semantics ($\mathcal{IF}, \mathit{ifield}, \mathit{inull}$) and an abstract domain $\mathcal{Val}^\sharp$. A variable `x` is abstracted in a flow-sensitive manner by an element $v \in \mathcal{Val}^\sharp$; a field f is abstracted in a flow-insensitive manner by a pair $(v_1, v_2) \in \mathcal{Val}^\sharp \times \mathcal{Val}^\sharp$ such that v_2 is the abstraction of `x.f` providing v_1 is the abstraction of `x`. The form of the abstract domain is defined by

$$\mathcal{Heap}^\sharp = \mathcal{F} \rightarrow \mathcal{Val}^\sharp \times \mathcal{Val}^\sharp \quad \mathcal{Env}^\sharp = \mathcal{Var} \rightarrow \mathcal{Val}^\sharp \qquad Abs = \mathcal{Heap}^\sharp \times (\mathcal{PP} \rightarrow \mathcal{Env}^\sharp)$$

The concretisation function $\gamma : Abs \rightarrow \mathcal{P}(\mathcal{State})$ is parametrised by γ_{null} and $\gamma_{\mathcal{L}}$ that are used to build the concretisation $\gamma_{\mathcal{Val}}$ of values. In the semantics, a value is either the constant *null* or a location l. The constant *null* can be

abstracted by any abstract value v part of $\gamma_{null} \subseteq \mathcal{Val}^\sharp$. As locations are abstraction of memory addresses in the semantics, a concretisation function $\gamma_{\mathcal{L}} : \mathcal{Val}^\sharp \to \mathcal{P}(\mathcal{L})$ would make little sense. The purpose of $\gamma_{\mathcal{L}} : \mathcal{Val}^\sharp \to \mathcal{P}(\mathcal{Class} \times \mathcal{F} \times \mathcal{IF})$ is to relate in the heap the class of the location and the instrumentation of the fields. As a result $\gamma_{\mathcal{Val}}$ is parametrised by a heap h and is defined as follows:

$$\frac{v^\sharp \in \gamma_{null}}{null \in \gamma^h_{\mathcal{Val}}(v^\sharp)} \qquad \frac{h(l) = (c, o) \qquad \forall f \in c.(c, f, o(f)_2) \in \gamma_{\mathcal{L}}(v^\sharp)}{l \in \gamma^h_{\mathcal{Val}}(v^\sharp)}$$

The abstraction of environments is defined component-wise, *i.e.,* the abstraction of each variable is non-relational.

$$\frac{\forall x, e(x) \in \gamma^h_{\mathcal{Val}}(e^\sharp(x))}{e \in \gamma^h_{\mathcal{Env}}(e^\sharp)}$$

The abstraction of the heap is also non-relational and each field is abstracted by a pair of abstract values.

$$\frac{\forall l, c, o.h(l) = (c, o) \Rightarrow (\forall f \in c.(c, f, o(f)_2) \in \gamma_{\mathcal{L}}(h^\sharp(f)_1) \Rightarrow o(f)_1 \in \gamma^h_{\mathcal{Val}}(h^\sharp(f)_2))}{h \in \gamma_{\mathcal{Heap}}(h^\sharp)}$$

Finally, the abstract domain Abs is a set of pairs of an abstract heap $h^\sharp$ and a flow-sensitive abstract environment $e^\sharp_{fs} : \mathcal{PP} \to \mathcal{Env}^\sharp$.

$$\frac{e \in \gamma^h_{\mathcal{Env}}\left(e^\sharp_{fs}(p)\right) \qquad h \in \gamma_{\mathcal{Heap}}\left(h^\sharp\right)}{(e, h, p) \in \gamma\left(h^\sharp, e^\sharp_{fs}\right)}$$

In the rest of this section, we model well-known analyses: Java byte-code verification [30] and a null-pointer analysis *à la* Fähndrich and Leino [20].

3.2 Bytecode Verification

For our core language, the purpose of byte-code verification consists in ensuring the absence of *LookupFail* errors. This error cannot be triggered if for every call instruction $\mathtt{x} := \mathtt{y.c_0.m(a_1, a_2)}$ the class of y is a subclass of $\mathtt{c_0}$. To rule out this error, byte-code verification would compute as abstraction for y a class c that is a subclass of $\mathtt{c_0}$. Byte-code verification does not require any instrumentation of the semantics. An abstract value $v \in \mathcal{Val}^\sharp = \mathcal{Class}_\perp$ is either a class c which represents either *null* or any object of class $c' \preceq c$, or $\perp$ which represents *null*.

$$\mathcal{IF} = \{\perp\} \quad inull = \perp \quad ifield(i) = \perp \quad \gamma_{null} = \mathcal{Val}^\sharp \quad \gamma_{\mathcal{L}}(c) = \{(c', f, i) \mid c' \preceq c\}$$

3.3 Null Pointer Analysis

Our parametrised concretisation can also model more sophisticated analyses similar to the null-pointer analysis of Fähndrich and Leino [20]. A key insight of

the analysis is the notion of *raw* type: an object of type $raw(c)$ is such that all the fields of c (or inherited from super-classes) are initialised. The crux is that the flow-insensitive abstraction of the heap is only valid for initialised fields. Hubert *et al.,* have formalised Fähndrich and Leino's type system in the context of abstract interpretation [25]. In order to track down the initialisation state of fields, they are using an instrumented semantics which annotates field with the status *def* (defined) as soon as their are initialised. With our semantics, this behaviour is modelled by the following instrumentation: $\mathit{I\mathcal{F}} = \{\mathit{def}, \mathit{undef}\} \qquad \mathit{inull} = \mathit{undef} \qquad \mathit{ifield}(i) = \mathit{def}\,.$

For precision, the analysis requires `this` to be given a more precise abstraction than other variables. Instead of a *raw* type, `this` is abstracted by an explicit mapping $f \in \mathcal{F} \to \{\mathit{Def}, \mathit{UnDef}\}$ where *Def* means *definitively defined* and *UnDef* means *may be defined*. In our framework, all the variables are treated in an homogeneous way and doing a special case for `this` is not possible. As a result, in our abstraction, all the variables are treated like `this`. This is a generalisation as a raw type $raw(c)$ is just a compact representation for $\lambda f.$ if $f \in c$ then *Def* else *UnDef*.

Another deviation from Fähndrich and Leino or Hubert *et al.,* is that our `new` statement is just allocating memory but does not calls a constructor. To precisely track down the state of a newly created object of class c, we introduce the type $\hat{c}$ which represents a totally uninitialised object of class exactly c.

$$\mathit{I\mathcal{F}}^\sharp = \{\mathit{Def}, \mathit{UnDef}\} \qquad \mathcal{V}\mathit{al}^\sharp = \{\mathit{MaybeNull}, \mathit{NotNull}\} \cup \widehat{\mathcal{C}\mathit{lass}} \cup (\mathcal{F} \to \mathit{I\mathcal{F}}^\sharp)$$

The type *MaybeNull* represents an arbitrary value and *NotNull* represents a non-null object with all its fields initialised. The type $\hat{c}$ represents an uninitialised object of class (exactly) c and a mapping $F \in \mathcal{F} \to \mathit{I\mathcal{F}}^\sharp$ represents an object such that the initialisation state of a field f is given by $F(f)$.

$$\gamma_{\mathit{I\mathcal{F}}}(\mathit{Def}) = \{\mathit{def}\} \qquad \gamma_{\mathit{I\mathcal{F}}}(\mathit{UnDef}) = \mathit{I\mathcal{F}} \qquad \gamma_{null} = \{\mathit{MaybeNull}\}$$

$$\gamma_{\mathcal{L}}(\mathit{MaybeNull}) = \mathcal{C}\mathit{lass} \times \mathcal{F} \times \mathit{I\mathcal{F}} \quad \gamma_{\mathcal{L}}(\mathit{NotNull}) = \{(c, f, v) \mid f \in c \Rightarrow v = \mathit{def}\}$$

$$\gamma_{\mathcal{L}}(\hat{c}) = \{c\} \times \mathcal{F} \times \mathit{I\mathcal{F}} \qquad \gamma_{\mathcal{L}}(F) = \{(c, f, v) \mid f \in c \Rightarrow v \in \gamma_{\mathit{I\mathcal{F}}}(F(f))\}$$

A feature of this analysis is that the abstraction of the heap is only valid for initialised field. This property is obtained as soon as an abstract heap $h^\sharp \in \mathcal{H}\mathit{eap}^\sharp$ is such that $h^\sharp(f)_1(f) = \mathit{def}$.

4 Generating Tractable Verification Conditions

The verification conditions generated for our restricted class of analyses are not automatically discharged by off-the-shelf provers. A significant difficulty is that the formulae quantify over the (infinite) set of memory locations and do not fall into known decidable fragments. To tackle this problem, our approach consists in generating abstract verification conditions that are geared towards the family of parametrised analyses presented in Section 3.1.

4.1 Almost Effectively Propositional Logic

The EPR logic, also known as the Bernays-Schönfinkel-Ramsey (BSR) class, is a decidable fragment of first-order logic where formulae are of the form

$$\exists^*\forall^*.\phi$$

where ϕ is a quantifier-free formula without function symbols. Piskac *et al.* [35] have shown how to decide EPR formulae extended with equality using the SMT solver Z3. Fontaine [23] has shown that the BSR class can be combined with decidable theories under mild assumptions (more relaxed than the standard Nelson-Oppen combination scheme). This makes this logic a good target for our verification conditions.

After transformation, our optimised verification conditions are of the form

$$\forall \bar{c} \in \mathcal{Class}, \bar{f} \in \mathcal{F}, \bar{i} \in \mathit{IF}, \bar{v} \in \mathcal{Var}.\phi$$

where $\bar{c}, \bar{f}, \bar{i}, \bar{v}$ are vectors of universally quantified variables and ϕ is a quantifier-free propositional formula built over the following atomic propositions

$$p ::= v^\sharp \in \gamma_{null} \mid (c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp) \mid c \preceq c \mid f \in c.$$

Here, $v^\sharp$ is a constant of the abstract domain $\mathcal{Val}^\sharp$; c is either a constant class name or a class variable bound in $\bar{c}$; f is either a constant field or a field variable bound in $\bar{f}$; i is an annotation of the form $\mathit{ifield}^n(i)$ where i is either a constant annotation or an annotation variable bound in $\bar{i}$; x is a variable.

In those formulae, constants play the role of existential variables. Observe that ground formulae $c \preceq c'$ and $f \in c$ are syntactic properties of programs that can be evaluated. The subclass predicate $\preceq$ is defined as the reflexive transitive closure of the `extends` relation. Fixpoints, even in the restricted form of transitive closure, are not expressible in first-order logic and are therefore outside the EPR fragment. We sidestep the difficulty by tabulating the relation subclass. We also tabulate the fact that a field f belongs to a class c. The translation is quadratic in the worst case. However, in practise, class hierarchies are never very deep [37]. The remaining atoms are static analysis dependent. Therefore, the reducibility to EPR is a property of the static analysis that can be decided by just looking at the definitions of γ_{null} and $\gamma_{\mathcal{L}}$.

The byte-code verification logic is trivially reducible to EPR: The atomic formula $v^\sharp \in \gamma_{null}$ always holds because *null* belongs to any abstract element $v^\sharp$. Moreover, $(c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp)$ reduces to $c \preceq v^\sharp$.

The null-pointer logic is also reducible to EPR. The atomic formula $v^\sharp \in \gamma_{null}$ can always be evaluated; it holds if and only if $v^\sharp = \mathit{MaybeNull}$. Atomic formulae of the form $(c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp)$ can be encoded in EPR extended with the theory of equality and a F interpreted function.

$$\begin{array}{ll}
(c, f, i) \in \gamma_{\mathcal{L}}(\mathit{MaybeNull}) & \text{iff } \mathit{True} \\
(c, f, i) \in \gamma_{\mathcal{L}}(\mathit{NotNull}) & \text{iff } f \in c \Rightarrow v = \mathit{def} \\
(c, f, i) \in \gamma_{\mathcal{L}}(\hat{c}') & \text{iff } c' = c \\
(c, f, i) \in \gamma_{\mathcal{L}}(F) & \text{iff } f \in c \Rightarrow F(f) = \mathit{Def} \Rightarrow i = \mathit{def}
\end{array}$$

The theory of equality can be reduced to EPR [35] and is not a problem. In this specific case, the interpreted function F is known and defined over a finite domain. For a given F, the formula $F(f) = Def$ can therefore be expanded into a (finite) disjunction $\bigwedge_{F(f')=Def} f' = f$. The obtained formula lies within the required fragment.

For these restrictions to be of interest we must show that our verification conditions can be expressed in this fragment. This is far from evident and is proved in Section 4.2

4.2 Abstract Verification Conditions

We show how to obtain sound abstract verification conditions The essential property of the VCs is that they fall in the logic fragment identified in Section 4.1. The reduction has been formally proved in Coq and is available [7].

Our verification conditions require the instrumentation to be monotonic *w.r.t.* to the abstraction of location.

$$\forall v^\sharp, c, f, i.(c, f, i) \in \gamma_{\mathcal{L}}(v^\sharp) \Rightarrow (c, f, \mathit{ifield}(i)) \in \gamma_{\mathcal{L}}(v^\sharp)$$

This property has already been identified as being instrumental for coping with multi-threading [20]. In a sequential setting, it could be relaxed at the cost of introducing an additional quantification modelling the fact that, for instance, during a method call the instrumentation can be updated an arbitrary number of times.

The VCs given in Fig. 1 use the following short-hands.

$$v_1^\sharp \overset{\bullet}{\sqsubseteq} v_2^\sharp \quad \triangleq \bigwedge \begin{array}{l} v_1^\sharp \in \gamma_{null} \Rightarrow v_2^\sharp \in \gamma_{null} \\ \forall c, i, f \in c.(c, f, i) \in \gamma_{\mathcal{L}}(v_1^\sharp) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(v_2^\sharp) \end{array}$$

$$v_1^\sharp \overset{\bullet}{\sqcap} v_2^\sharp \overset{\bullet}{\sqsubseteq} v_3^\sharp \triangleq \bigwedge \begin{array}{l} v_1^\sharp \in \gamma_{null} \wedge v_2^\sharp \in \gamma_{null} \Rightarrow v_3^\sharp \in \gamma_{null} \\ \forall c, i, f \in c. \\ \qquad (c, f, i) \in \gamma_{\mathcal{L}}(v_1^\sharp) \wedge (c, f, i) \in \gamma_{\mathcal{L}}(v_2^\sharp) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(v_3^\sharp) \end{array}$$

Given an abstraction $(H, E) \in \mathit{Heap}^\sharp \times (\mathcal{PP} \to \mathcal{E}nv^\sharp)$ of the program, we generate for each program point p a verification condition $VC_p^{\sharp(H,E)}$ for the statement $s \in \mathit{Stmt}$ such that $\texttt{get\_stmt}(p) = s$. For each method signature $m' \in \mathit{Class} \times \mathit{Method}$ which overrides a method $m \in \mathit{Class} \times \mathit{Method}$ in a sub-class, we also generate abstract verification conditions $VC^{\sharp(H,E)}(m', m)$ modelling the usual variance/co-variance rules for method redefinitions. The comprehensive VCs are given in Fig. 1. In all rules, the terms of the statement on which the VC is produced are capital letters in a True-Type font (*e.g.*, `x`) and the two parts of the abstraction are written in italic capital letters. We do not indicate the sorts of the quantified variables to keep the formulae readable, but all v are variables in $\mathcal{V}ar$, c are classes in Class, f are fields in $\mathcal{F}$, i are instrumentations of fields in $\mathcal{IF}$, except in the VC for call instructions, where it is specified $\forall i \in \{0, 1\}$ to avoid repeating the condition.

$$VC^\sharp(\texttt{skip})^{(H,E)}_{cpp} = \forall v.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v)$$

12

$$VC^\sharp(\texttt{x} := \texttt{null})^{(H,E)}_{cpp} = \begin{cases} \forall v \neq \texttt{x}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge E(cpp^+)(\texttt{x}) \in \gamma_{null} \end{cases}$$

$$VC^\sharp(\texttt{x} := \texttt{x}')^{(H,E)}_{cpp} = \begin{cases} \forall v \neq \texttt{x}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge E(cpp)(\texttt{x}') \overset{\bullet}{\sqsubseteq} E(cpp^+)(\texttt{x}) \end{cases}$$

$$VC^\sharp(\texttt{x} := \texttt{y.f})^{(H,E)}_{cpp} = \begin{cases} \forall v \neq \texttt{x}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge \forall c, i. \\ \quad \texttt{f} \in c \Rightarrow \\ \quad (c, \texttt{f}, i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{y})) \Rightarrow \\ \quad ((c, \texttt{f}, i) \in \gamma_{\mathcal{L}}(H(\texttt{f})_1) \Rightarrow H(\texttt{f})_2 \in \gamma_{null}) \Rightarrow \\ \quad E(cpp^+)(\texttt{x}) \in \gamma_{null} \\ \wedge \forall c, c', f', i, i'. \\ \quad \texttt{f} \in c \Rightarrow \\ \quad (c, \texttt{f}, i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{y})) \Rightarrow \\ \quad ((c, \texttt{f}, i) \in \gamma_{\mathcal{L}}(H(\texttt{f})_1) \Rightarrow (c', f', i') \in \gamma_{\mathcal{L}}(H(\texttt{f})_2)) \Rightarrow \\ \quad (c', f', i') \in \gamma_{\mathcal{L}}(E(cpp^+)(\texttt{x})) \end{cases}$$

$$VC^\sharp(\texttt{x} := \texttt{new c})^{(H,E)}_{cpp} = \begin{cases} \forall v \neq \texttt{x}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge \forall f \in \texttt{c}.(\texttt{c}, f, inull) \in \gamma_{\mathcal{L}}(E(cpp^+)(x)) \\ \wedge \forall c', f \in c'.(c', f, inull) \in \gamma_{\mathcal{L}}(H(f)_1) \Rightarrow H(f)_2 \in \gamma_{null} \end{cases}$$

$$VC^\sharp(\texttt{x.f} := \texttt{y})^{(H,E)}_{cpp} = \begin{cases} \forall c, i. \\ \quad \texttt{f} \in c \Rightarrow \\ \quad (c, \texttt{f}, i) \in E(cpp)(x) \Rightarrow \\ \quad (c, \texttt{f}, ifield(i)) \in E(cpp^+)(x) \\ \wedge \forall v \neq \texttt{x}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge \forall i, c, f' \neq \texttt{f}. \\ \quad \texttt{f} \in c \Rightarrow \\ \quad (c, f', i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{x})) \\ \quad (c, f', i) \in \gamma_{\mathcal{L}}(E(cpp^+)(\texttt{x})) \\ \wedge E(cpp)(\texttt{y}) \overset{\bullet}{\sqsubseteq} H(\texttt{f})_2 \\ \wedge \forall c, f', i. \\ \quad (c, \texttt{f}, i) \in \gamma_{\mathcal{L}}(E(cpp)(x)) \Rightarrow (c, \texttt{f}, ifield(i)) \in \gamma_{\mathcal{L}}(H(f')_2) \end{cases}$$

$$VC^\sharp(\texttt{x} := \texttt{y.c.m}(\texttt{v}_0, \texttt{v}_1))^{(H,E)}_{cpp} = \begin{cases} \forall i, f, c' \preccurlyeq \texttt{c}.(c', f, i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{y})) \Rightarrow (c', f, v) \in \gamma_{\mathcal{L}}(E((\texttt{c}, \texttt{m})_0)(\texttt{this})) \\ \wedge \forall i \in \{0,1\}.E(cpp)(v_i) \overset{\bullet}{\sqsubseteq} E((\texttt{c}, \texttt{m})_0)(p_i) \\ \wedge E((\texttt{c}, \texttt{m})_\infty)(\texttt{res}) \overset{\bullet}{\sqsubseteq} E(cpp^+)(x) \\ \wedge \forall v \notin \{\texttt{x}, \texttt{y}, \texttt{v}_0, \texttt{v}_1\}.E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge E((\texttt{c}, \texttt{m})_\infty)(\texttt{this}) \overset{\bullet}{\sqcap} E(cpp)(y) \overset{\bullet}{\sqsubseteq} E(cpp^+)(y) \\ \forall i \in \{0,1\}.E((\texttt{c}, \texttt{m})_\infty)(\texttt{p}_\texttt{i}) \overset{\bullet}{\sqcap} E(cpp)(\texttt{v}_\texttt{i}) \overset{\bullet}{\sqsubseteq} E(cpp^+)(\texttt{v}_\texttt{i}) \end{cases}$$

$$VC^\sharp(\texttt{Jnull}(\texttt{x}, p')^{(H,E)}_{cpp} = \begin{cases} E(cpp)(\texttt{x}) \in \gamma_{null} \Rightarrow E(cpp^+(x)) \in \gamma_{null} \\ \wedge \forall v \neq \texttt{x}.E(cpp)(\texttt{x}) \in \gamma_{null} \Rightarrow E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(cpp^+)(v) \\ \wedge \forall c, i, f \in c.(c, f, i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{x})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(cpp^+)(\texttt{x})) \\ \wedge \forall c, i, f, v \neq x.(f \in c \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(cpp)(\texttt{x}))) \Rightarrow E(cpp)(v) \overset{\bullet}{\sqsubseteq} E(p')(v) \end{cases}$$

$$VC^{\sharp(H,E)}(m', m) = \begin{cases} E(m'_\infty)(\texttt{res}) \overset{\bullet}{\sqsubseteq} E(m_\infty)(\texttt{res}) \\ \wedge \forall c \preccurlyeq class(m'), f, i.(c, f, i) \in \gamma_{\mathcal{L}}(E(m_0)(\texttt{this})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(m'_0)(\texttt{this})) \\ \wedge \forall i \in \{0,1\}.E(m_0)(p_i) \overset{\bullet}{\sqsubseteq} E(m'_0)(p_i) \\ \wedge \forall v \notin \{\texttt{this}, \texttt{p}_0, \texttt{p}_1\}.E(m_0)(v) \in \gamma_{null} \end{cases}$$

Fig. 1. Optimised verification conditions

Assignments. We produce different VCs for assignments $\mathtt{x} := e$ depending on the expression e. If e is simply `null`, then the VC simply propagates the information on all variables different from `x` and checks that the abstract value for `x` at the next program point can represent a null value.

$$VC^\sharp(\mathtt{x} := \mathtt{null})^{(H,E)}_p = \begin{cases} \forall v \neq \mathtt{x}.\ E(p)(v) \overset{\bullet}{\sqsubseteq} E(p^+)(v) \\ \wedge E(p^+)(\mathtt{x}) \in \gamma_{null} \end{cases}$$

The other VC for assignment deals with instructions of the form $\mathtt{x} := \mathtt{x}'$, and checks that the information on $\mathtt{x}'$ are propagated to the information on `x` at the next program point, replacing the condition $E(p^+)(\mathtt{x}) \in \gamma_{null}$ by $E(p)(\mathtt{x}') \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{x})$.

Method calls. Along the same lines, most of the conditions of the VC for call statements $x := \mathtt{y.C.M}(\mathtt{V}_0, \mathtt{V}_1)$ simply check that the correct information is propagated. First, the information on all local variables that are not concerned by the call—variables that are neither $\mathtt{x}, \mathtt{y}, \mathtt{V}_0$ nor $\mathtt{V}_1$—must be propagated to the next program point.

$$\forall v \notin \{\mathtt{x}, \mathtt{y}, \mathtt{V_0}, \mathtt{V_1}\}.\ E(p)(v) \overset{\bullet}{\sqsubseteq} E(p^+)(v)$$

Then, the VC must check that the pre-condition of the method called is enforced, *i.e.*, it must check that the information on the argument of the call $\mathtt{y}$,$\mathtt{V}_0$ and $\mathtt{V}_1$ implies the information on the parameter `this`,p_0 and p_1 at the entry point of the method. We take the entry point of the implementation of the method in the highest possible class $(\mathtt{C}, \mathtt{M})_0$. A different VC checks that all implementations respect the usual variance/co-variance rule for method redefinitions.

$$\forall i, f, c' \preccurlyeq \mathtt{C}.\ (c', f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{y})) \Rightarrow (c', f, v) \in \gamma_{\mathcal{L}}(E((\mathtt{C}, \mathtt{M})_0)(\mathtt{this}))$$
$$\forall i \in \{0, 1\}.\ E(p)(\mathtt{V}_i) \overset{\bullet}{\sqsubseteq} E((\mathtt{C}, \mathtt{M})_0)(p_i)$$

The constraint concerning the parameter `this` is a bit relaxed: we know that the object `y` is not null and at most of class `C`. A different VC is in charge of checking that the lookup never fails. The VC checks that the information on `y` at the call point *up to* `C` is propagated to the information on `this` at the entry point.

Finally, the VC checks that the information at the exit point $(\mathtt{C}, \mathtt{M})_\infty$—*i.e.*, the post-condition of the method—is propagated.

$$E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{res}) \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{x})$$
$$E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{this}) \overset{\bullet}{\sqcap} E(p)(\mathtt{y}) \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{y})$$
$$\forall i \in \{0, 1\}.\ E((\mathtt{C}, \mathtt{M})_\infty)(\mathtt{p_i}) \overset{\bullet}{\sqcap} E(p)(\mathtt{V}_i) \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{V}_i)$$

Note that even if the semantics specify that the value—*i.e.*, the location—of the variables `y`, $\mathtt{V}_0$ and $\mathtt{V}_1$ is not touched by the call, the object they point to

may have been modified by the call, *e.g.*, more fields could be initiated. Therefore, the information at the next program point on these variables is actually the intersection of the information at the call point—*i.e.*, $E(p)(\bullet)$—and of the information on the parameters—`this` for `y`, $\mathtt{p}_0$ for $\mathtt{V}_0$ and $\mathtt{p}_1$ for $\mathtt{V}_1$—at the exit point of the method $E((\mathtt{C},\mathtt{M})_\infty)(\bullet)$, hence the use of the shorthand $v_1^\sharp \overset{\bullet}{\sqcap} v_2^\sharp \overset{\bullet}{\sqsubseteq} v_3^\sharp$.

Conditional tests. A program point p annotated with a branching statement $\mathtt{Jnull}(\mathtt{x}, p')$ generates one VC, with conditions related to the two branches. If the information on `x` at program point p indicates that the variable can be null, *i.e.*, $E(p)(\mathtt{x}) \in \gamma_{null}$, then the jump may occur, therefore the information on `x` at p' must signal that `x` may be null, and the information on all other variables must be propagated from p to p'.

$$E(p)(\mathtt{x}) \in \gamma_{null} \Rightarrow E(p')(\mathtt{x}) \in \gamma_{null}$$
$$\forall v \neq \mathtt{x}.\ E(p)(\mathtt{x}) \in \gamma_{null} \Rightarrow E(p)(v) \overset{\bullet}{\sqsubseteq} E(p')(v)$$

As soon as the information on `x` at p indicates that the variable can be not-null, *i.e.*, if $(c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{x}))$ is true, then some executions may continue to p^+ and the information must be propagated accordingly.

$$\forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{x})) \Rightarrow (c, f, i) \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{x}))$$
$$\forall v \neq \mathtt{x}, \forall c, i, f \in c.\ (c, f, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{x})) \Rightarrow E(p)(v) \overset{\bullet}{\sqsubseteq} E(p^+)(v)$$

Note that the information that `x` may be null at p is not propagated to p^+, we use a constraint a bit more relaxed than a simple $E(p)(\mathtt{x}) \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{x})$, and can therefore certify *guard-sensitive* analyses *i.e.*, exploit guards to refine analysis results.

Object allocation. The VC for the $\mathtt{x} \coloneqq \mathtt{new\ C}$ statement is straightforward. It only has to check—besides the fact that variables other than `x` are unchanged—that the information on `x` at the next program point can account for the fact that all the fields of the object stored in `x` have a *inull* annotation and have a null value.

$$\forall f \in \mathtt{C}.\ (\mathtt{C}, f, inull) \in \gamma_{\mathcal{L}}(E(p^+)(x))$$
$$\forall f \in \mathtt{C}.\ (\mathtt{C}, f, inull) \in \gamma_{\mathcal{L}}(H(f)_1) \Rightarrow H(f)_2 \in \gamma_{null}$$

Accesses in the heap. The VC for a program point p annotated with an access in the heap $\mathtt{x} \coloneqq \mathtt{y.f}$ states that the information on `f` in the flow-insensitive abstraction of the heap, *i.e.*, $H(\mathtt{f})_2$, should be propagated to the information on `x` at the next program point. Nonetheless, recall that the abstraction of the heap may distinguish between the possible annotations of `f`. Therefore, the information from H must be propagated to $E(p^+)(\mathtt{x})$ depending on what the information

on $\mathtt{y}$ at p can say about the flag on $\mathtt{y.f}$.

$$
\begin{array}{l}
\forall c, i. \\
\quad \mathtt{f} \in c \Rightarrow \\
\quad (c, \mathtt{f}, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{y})) \Rightarrow \\
\quad \Big((c, \mathtt{f}, i) \in \gamma_{\mathcal{L}}(H(\mathtt{f})_1) \Rightarrow H(\mathtt{f})_2 \in \gamma_{null}\Big) \Rightarrow \\
\quad E(p^+)(\mathtt{x}) \in \gamma_{null} \\
\forall c, c', f', i, i'. \\
\quad \mathtt{f} \in c \Rightarrow \\
\quad (c, \mathtt{f}, i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{y})) \Rightarrow \\
\quad \Big((c, \mathtt{f}, i) \in \gamma_{\mathcal{L}}(H(\mathtt{f})_1) \Rightarrow (c', f', i') \in \gamma_{\mathcal{L}}(H(\mathtt{f})_2)\Big) \Rightarrow \\
\quad (c', f', i') \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{x}))
\end{array}
$$

There are two kinds of information to propagate: f may be null, *i.e.*, $H(\mathtt{f})_2 \in \gamma_{null}$, and the set of objects the abstraction of f may correspond to, hence the two conditions. Remark that the parentheses does not allows the use of the shorthand $H(\mathtt{f})_2 \overset{\bullet}{\sqsubseteq} E(p^+)(\mathtt{x})$.

Updates in the heap. The VC for a program point p annotated with an update in the heap $\mathtt{x.f} \coloneqq \mathtt{y}$ checks that the information on $\mathtt{y}$ is propagated in the heap

$$E(p)(\mathtt{y}) \overset{\bullet}{\sqsubseteq} H(\mathtt{f})_2$$

but must also checks that the abstraction accounts for the update on the flag attached to the field. It must be accounted for in the abstraction of the environment, for all objects in which $\mathtt{f}$ is defined, but only for the field $\mathtt{f}$

$$
\begin{array}{l}
\forall c, i. \\
\quad \mathtt{f} \in c \Rightarrow \\
\quad (c, \mathtt{f}, i) \in E(p)(x) \Rightarrow \\
\quad (c, \mathtt{f}, ifield(i)) \in E(p^+)(x) \\
\forall i, c, f' \neq \mathtt{f}. \\
\quad \mathtt{f} \in c \Rightarrow \\
\quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p)(\mathtt{x})) \Rightarrow \\
\quad (c, f', i) \in \gamma_{\mathcal{L}}(E(p^+)(\mathtt{x}))
\end{array}
$$

and it must be accounted for in the heap.

$$\forall c, f', i. \quad (c, \mathtt{f}, i) \in \gamma_{\mathcal{L}}(E(p)(x)) \Rightarrow (c, \mathtt{f}, ifield(i)) \in \gamma_{\mathcal{L}}(H(f')_2)$$

Theorem 1. *Let P be a program and (H, E) be the untrusted result of an analysis such that the instrumentation is monotonic. If the abstract VCs hold for all the statements ($\forall p \in \mathcal{PP}, s \in \mathcal{S}tmt.\mathtt{get\_stmt}(p) = s \Rightarrow VC^{\sharp^{(H,E)}}_{p}(s)$) and the abstract VCs hold for method redefinitions ($\forall m, m'.override(m', m) \Rightarrow VC^{\sharp^{(H,E)}}(m', m)$) then the concrete VCs hold and as a consequence the analysis result is sound ($\mathcal{R}each \subseteq \gamma(H, E)$).*

This theorem is proved correct in our Coq development [7].

For each statement $s \in \mathcal{S}tmt$ at program point p which can potentially be responsible for an error $e \in \mathcal{E}rr$ we generate an abstract verification condition $Chk^\sharp(s)_p^{(H,E)}$ ruling out this error.

$$Chk^\sharp(\mathtt{x} := \mathtt{y.f})_p^{(H,E)} = \neg(E(p)(\mathtt{y}) \in \gamma_{null})$$
$$Chk^\sharp(\mathtt{x.f} := \mathtt{y})_p^{(H,E)} = \neg(E(p)(\mathtt{x}) \in \gamma_{null})$$
$$Chk^\sharp(\mathtt{x} := \mathtt{y.c.m}(\mathtt{v_0}, \mathtt{v_1}))_p^{(H,E)} = \begin{cases} \neg(E(p)(\mathtt{y}) \in \gamma_{null}) \\ \wedge \forall c', f, i.(c', f, i) \in E(p)(\mathtt{y}) \Rightarrow c' \preceq \mathtt{c} \end{cases}$$

Theorem 2. *Let P be a program and (H, E) be a sound analysis ($\mathcal{R}each \subseteq \gamma(H, E)$). If the abstract VCs hold for all the statements*

$$\forall p \in \mathcal{PP}, s \in \mathcal{S}tmt.\mathtt{get\_stmt}(p) = s \Rightarrow Chk^{\sharp\,(H,E)}_p(s)$$

then the absence of errors is guaranteed by the static analysis result ($\forall s, e.s \in \mathcal{R}each \Rightarrow s \not\rightsquigarrow e$).

5 Experiments

For our experiments, we consider the null pointer analysis presented in Section 3.3. To get type annotations, we have ported the NIT implementation [25] to the Sawja platform [24] thus benefiting from a Bytecode Intermediate Language [19] that is closed to the idealised language of Section 2. For the time being, we do not generate VCs for BIR instructions that do not have a direct counterpart in our idealised language. In particular, we ignore instructions manipulating primitive types, static fields and static methods (with the notable exception of constructors). Following Fähndrich and Leino [20], a constructor implicitly defines all the fields of the current class. We emulate this behaviour by adapting our VC for method returns. For a constructor, we add in the hypotheses that the fields of the current class are necessarily defined.

For other instructions, we generate VCs according to Figure 1. At generation time, verification conditions are partially evaluated with respect to the analysis result. In particular, this is the case for terms of the form $\gamma_L(E(\mathtt{p})(\mathtt{v}), (c, f, i))$ where E, $\mathtt{p}$ and $\mathtt{v}$ are constant or of the form $\gamma_L(E(\mathtt{p})(v), (c, f, i))$ where only E and $\mathtt{p}$ are constant. The generated VCs are then processed by Why3 and sent to different ATPs.

Results: All experiments [7] were done on a laptop running Linux with 4GB memory and Intel Core 2 Duo cpu at 2.93GHz. We used Why3 0.80 version and tested the ability of different provers to discharge the VCs. We limited our choice to provers to which Why3 could interface to *off-the-shelf*, and not, for instance, the latest winners of the EPR category of the CASC competition [36]. We generated VCs for a limited set of small Java programs, testing the different cases of the analysis. All VCs were discharged in less than 0.2 seconds, most

in less than 0.05 seconds, times hardly significant. Some provers were not able to discharge all VCs, TPTP provers in particular did not managed to discharge most VCs. Among SMT provers, only CVC3 was able to discharge all VCs, alt-ergo failing very quickly in some cases, and Z3 reaching timeout (5 seconds). This differences could be due to the encoding we used, and more experiment would be needed to understand why some provers performed better.

Nevertheless, our experiments showed that the VC calculus presented in Section 4.2 produced VCs consistently discharged by multiple provers, therefore demonstrating the relevance of the EPR fragment presented in Section 4.1 as a framework for efficient result certification of object-oriented properties. The only limiting factor to scalability appears to be the encoding of the class hierarchy $c' \preccurlyeq c$ and the relation $f \in c$. Analysing programs that use the standard Java library may involve hundreds of classes and thousands of fields, and describing an efficient encoding for relation on such large domains is a problem in its own right.

6 Related Work

Static program analysis is a core technology for verifying software. Most static analysers are complex pieces of software that are not without errors. Hence, we have witnessed a growing interest in *certified static analysis*.

Certified static analysis has been pioneered by Necula in his seminal work on Proof-Carrying Code (PCC) [32]. Necula insists on the fact that proof generation should be automatic and therefore invariant generation should be based on static analysis. The back-end of a PCC architecture is a proof generating theorem prover able to discharge the verification conditions. In a PCC setting, the Touchstone theorem prover [33] generates LF proof terms for a quantifier-free fragment of first-order logic enhanced with a specific theory for modeling types and typing rules. For the family of static analyses we consider, a traditional VC-Gen would not generate verification conditions in the scope of Touchstone. The Open Verifier [14] aims at providing a generic Proof-Carrying architecture for proving memory safety. For each analysis, a new type checker is an untrusted module which, using a scripting language, instructs the kernel of a proof strategy for discharging the verification conditions. In our work, the verification conditions are compiled for our family of analyses and the verification conditions are discharged using trusted solvers.

Foundational Proof-Carrying Code [2] proposes to reduce the TCB to a proof-checker and the definition of the program semantics. A foundational proof of safety for a static analyser can be obtained by certifying the analyser inside a proof-checker. Klein and Nipkow have formalised the Java bytecode verifier in Isabelle [27]. Pichardie *et al.* [13,34] formalised the abstract interpretation framework [17] in Coq and used it to prove the soundness of several program analysers. This approach requires to develop and prove in Coq the whole analyser which is a formidable effort of certification and raises efficiency concerns, Coq being a pure lambda-calculus language. Another way to obtain a founda-

tional proof of safety is to certify, inside the proof-checker, a verifier of analysis result rather than the analyser. Besson *et al.* [9] applied this *result certification methodology* [38] to a polyhedral analysis, developing an analyser together with a dedicated checker whose soundness is proved inside Coq. These works target a single analysis but aim at a minimal TCB. Our approach is more automatic and capture a family of analyses at the cost of integrating provers in the TCB. However, generating foundational proofs for provers is also an active research area [11,8,3]. These works pave the way for foundational proofs of family of static analysis results.

Albert *et al.* [1] who have shown how results of the state-of-the-art static analysis system COSTA can be checked using the verification tool Key. COSTA produces guarantees on how resources are used in programs. Resource guarantees are expressed as upper bounds on number of iterations and worst-case estimation of resource usage, and injected into Key as JML annotations. The derived proof obligations are proved automatically using the prover of Key.

7 Conclusion and Further Work

Result verification of abstract interpretation-based static analysers can be implemented using ATP, by injecting the static analysis results into a program verification tool, and generating the corresponding verification conditions. A straightforward generation from the operational semantics will generate VCs that are likely to be too complex for current provers. For this approach to work, the verification conditions must be generated with care. We show how to generate VCs optimised for a class of analyses (here including byte code verification and null pointer analysis) and which fall in a logical fragment that is amenable to automatic proving. We have conducted a machine-checked proof (in Coq) that these VCs are sound with respect to the standard VCs for the semantics. This approach has been validated through an implementation with the Why3 tool which is capable of verifying analysis results in a few seconds using off-the-shelf solvers.

Further work includes larger experiments for assessing the scalability of our approach. We are confident that our EPR VCs are *easy* instances (quantifications can be bounded by exploiting the class hierarchy) that will be discharged without problem by off-the-shelf provers. However, this needs to be validated experimentally. We also intend to widen the family of analyses in the scope of our approach and study how to extend the class studied in the present paper with relational numeric analyses. We expect the VCs to fall in a combination of EPR with arithmetic. A longer-term research goal consists in automating the generation of provably sound tractable VCs. Recently, Marché and Tafat have shown how to prove a *classic* WP calculus in Why3 [31]. We will investigate how to adapt this approach for custom Verification Condition Generators specialised for classes of static analyses.

Acknowledgments. Thanks are due to P. Vittet for porting the Null Inference Tool (Nit) to Sawja and helping with the experiments.

References

1. E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *PEPM '11*, SIGPLAN, pages 73–76. ACM, 2011.
2. A. W. Appel. Foundational proof-carrying code. In *LICS 2001*, pages 247–256. IEEE Computer Society, 2001.
3. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *CPP'11*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.
4. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
6. C. Barrett and C. Tinelli. Cvc3. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
7. F. Besson, P.-E. Cornilleau, and T. Jensen. Why3 and Coq source of the development, 2012. available at `http://www.irisa.fr/celtique/ext/chk-sa`.
8. F. Besson, P.-E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In *CPP'11*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011.
9. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *TGC'10*, volume 6084 of *LNCS*, pages 253–267. Springer, 2010.
10. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011*, pages 53–64, 2011.
11. S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *IJCAR'10*, volume 6173 of *LNCS*, pages 107–121. Springer, 2010.
12. T. Bouton, D. C. B. d. Oliveira, D. Déharbe, and P. Fontaine. VeriT: an open, trustable and efficient SMT-solver. In *CADE'09*, LNCS, pages 151–156. Springer, 2009.
13. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.
14. B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The Open Verifier framework for foundational verifiers. In *TLDI'05*, pages 1–12. ACM, 2005.
15. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
16. P.-E. Cornilleau. Prototyping static analysis certification using Why3. In *Boogie 2012*, 2012.
17. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
18. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

19. D. Demange, T. P. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for java bytecode. In *APLAS*, pages 97–113, 2010.
20. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. *OOPSLA'03*, pages 302–312, 2003.
21. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.
22. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245. ACM, 2002.
23. P. Fontaine. Combinations of Theories and the Bernays-Schönfinkel-Ramsey Class. In *VERIFY*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
24. L. Hubert, N. Barré, F. Besson, D. Demange, T. P. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static analysis workshop for java. In *FoVeOOS*, pages 92–106, 2010.
25. L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *FMOODS'08*, volume 5051 of *LNCS*, pages 132–149. Springer, 2008.
26. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.
27. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.
28. X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.
29. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM, 2006.
30. T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.
31. C. Marché and A. Tafat. Weakest Precondition Calculus, Revisited using Why3. Research report RR-8185, INRIA, Dec. 2012.
32. G. C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM, 1997.
33. G. C. Necula and P. Lee. Proof generation in the Touchstone theorem prover. In *CADE'00*, volume 1831 of *LNCS*, pages 25–44. Springer, 2000.
34. D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. in French.
35. R. Piskac, L. M. de Moura, and N. Bjørner. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.
36. G. Sutcliffe. The 5th IJCAR Automated Theorem Proving System Competition - CASC-J5. *AI Communications*, 24(1):75–89, 2011.
37. E. D. Tempero, J. Noble, and H. Melton. How do java programs use inheritance? an empirical study of inheritance in java software. In *ECOOP'08*, volume 5142 of *LNCS*, pages 667–691. Springer, 2008.
38. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

A Operational Semantics

The semantic rules can be found in Fig. 2 and Fig. 3.

$$\text{Skip}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{skip}}{s \longrightarrow (s.env, s.hp, s.cpp^{+})}$$

$$\text{Assign}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{e} \quad x\ \textit{is assignable} \quad (s.env, e) \Rightarrow v}{s \longrightarrow (s.env[x \leftarrow v]\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{JumpNull}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{Ifnull}(\texttt{t}, \texttt{p}') \quad (s.env, t) \Rightarrow \mathit{null}}{s \longrightarrow (s.env\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{JumpLoc}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{Ifnull}(\texttt{t}, \texttt{p}') \quad (s.env, t) \Rightarrow l}{s \longrightarrow (s.env\ ,\ s.hp\ ,\ p')}$$

$$\text{New}\ \frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = x := \texttt{new c} \quad x\ \textit{is assignable} \\ s.hp[l] = \bot \qquad o = \lambda f.(\mathit{null}, \mathit{inull})\end{array}}{s \longrightarrow (s.env[x \leftarrow l]\ ,\ s.hp[l \leftarrow (c, o)]\ ,\ s.cpp^{+})}$$

$$\text{Getfield}\ \frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.f} \quad x\ \textit{is assignable} \\ s.env[y] = l \quad s.hp[l] = (c, o) \quad o[f] = (v, i)\end{array}}{s \longrightarrow (s.env[x \leftarrow v]\ ,\ s.hp\ ,\ s.cpp^{+})}$$

$$\text{Putfield}\ \frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = \texttt{x.f} := \texttt{y} \qquad s.env[x] = l \qquad s.hp[l] = (c, o) \\ v' = s.env[y] \quad (v, i) = o[f] \quad i' = \mathit{ifield}(i) \quad o' = o[f \leftarrow (v', i')]\end{array}}{s \longrightarrow (s.env\ ,\ s.hp[l \leftarrow (c, o')]\ ,\ s.cpp^{+})}$$

$$\text{Call}\ \frac{\begin{array}{c}\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x\ \textit{is assignable} \\ s \rhd_{c_0, m} (\mathit{init}, p_{end}) \quad \mathit{init} \longrightarrow^{*} \mathit{end} \quad \mathit{end.cpp} = p_{end}\end{array}}{s \longrightarrow (s.env[x \leftarrow \mathit{end.env}[\texttt{res}]]\ ,\ \mathit{end.hp}\ ,\ s.cpp^{+})}$$

$$\text{SCall}\ \frac{\begin{array}{c}s.env[y] = l \quad s.hp[l] = (c, o) \quad (s.env, a_0) \Rightarrow v_0 \quad (s.env, a_1) \Rightarrow v_1 \\ \texttt{lookup}(c)(c_0, m) = c' \qquad \texttt{sig}(c', m) = (p_{beg}, p_{end}) \\ env' = (\lambda x.\mathit{null})[\texttt{this} \leftarrow l][\texttt{p}_0 \leftarrow v_0][\texttt{p}_1 \leftarrow v_1]\end{array}}{s \rhd_{c_0, m} ((env'\ ,\ s.hp\ ,\ p_{beg}), p_{end})}$$

Fig. 2. Semantics

$$\text{GetfieldNullP}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.f} \quad x \textit{ is assignable} \quad s.env[y] = null}{s \rightsquigarrow \mathit{NullPointer}}$$

$$\text{PutfieldNullP}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x.f} := \texttt{y} \quad x \textit{ is assignable} \quad s.env[x] = null}{s \rightsquigarrow \mathit{NullPointer}}$$

$$\text{CallNullP}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \quad s.env[y] = null}{s \rightsquigarrow \mathit{NullPointer}}$$

$$\text{LookupFail}\ \frac{\texttt{get\_stmt}(s.cpp) = \texttt{x} := \texttt{y.c}_0\texttt{.m}(\texttt{a}_0, \texttt{a}_1) \quad x \textit{ is assignable} \quad s.env[y] = l \quad s.hp[l] = (c, o) \quad \texttt{lookup}(c)(c_0, m) = \bot}{s \rightsquigarrow \mathit{LookupFail}}$$

Fig. 3. Error conditions